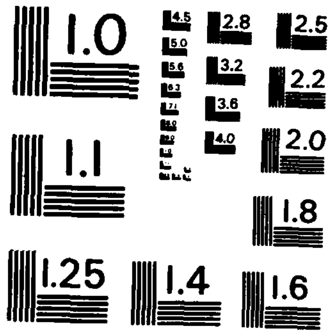
AD-A160 135 ON USING INVERTED TREES FOR UPDATING GRAPH PROPERTIES (U) MARYLAND UNIV COLLEGE PARK CENTER FOR AUTOMATION RESEARCH S PAWAGI ET AL. MAY 85 CAR-TR-117 1/1

UNCLASSIFIED AFOSR-TR-85-0822 F49620-83-C-0082 F/G 12/1 NL

END

FILMED

DTIC


1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

2

May 1985

CAR-TR-117
CS-TR-1492

On Using Inverted Trees for Updating Graph Properties

Shaunak Pawagi
I. V. Ramakrishnan
Department of computer Science
University of Maryland
College Park, MD 20742

COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

CAR-TR-117
CS-TR-1492

May 1985

On Using Inverted Trees for Updating Graph Properties

Shaunak Pawagi
I. V. Ramakrishnan

Department of computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

Fast parallel algorithms are presented for updating connected components and bridges of an undirected graph when a minor change has been made to the graph, such as addition or deletion of vertices and edges. The machine model used is a parallel random access machine which allows simultaneous reads but prohibits simultaneous writes into the same memory location. The algorithms described in this paper require O(log n) time and use $O(n^2)$ processors. These algorithms are efficient when compared to previously known algorithms for finding connected components and bridges that require $O(\log^2 n)$ time and use $O(n^2)$ processors. The previous solution is maintained using an inverted tree (a rooted tree where a node points towards its parent) and after a minor change the new solution is rapidly computed from this tree.

The support of the first author by the Air Force Office of Scientific Research under Contract F-49620-83-C-0082, and of the second author by the Office of Naval Research under Contract N00014-84-K-0530, and by the National Science Foundation under grant ECS-84-04399, is gratefully acknowledged.

1. Introduction

Efficient algorithms for a variety of graph problems have been developed in past (see [2] for an extensive bibliography). However, one aspect of this algorithmic approach to graph theory has not been dealt with extensively. This aspect is concerned with recomputing the properties of a graph after *incremental* changes have been made to it such as addition and deletion of edges and vertices of the graph. Such recomputations are also referred to as *updating* graph properties. Incremental changes made to a graph model dynamic behavior of the underlying system that it represents. If such incremental changes are minor (such as deletion and addition of an edge or vertex) then it should be possible to construct efficient algorithms to recompute the properties of the graph when compared to algorithms that do not use any of the previous information. Such algorithms that make use of the previous solution (and possibly some additional information) are termed *incremental* algorithms, while the algorithms for initial computation of graph properties are referred to as *start-over* algorithms in [1]. The kinds of minor modifications that are considered here are as follows. First, a vertex may be added along with the edges incident on it. Second, an individual edge may be deleted or added.

We can characterize incremental graph algorithms in terms of stages. The first stage is to determine what part of the solution is unaffected by the graph change. This is important as substantial gains can be made by avoiding the recomputation of the unaffected part of the solution. The second stage is the actual recomputation of that part of the solution which is affected by the minor graph change. This stage can be implemented efficiently by making use of the previous solution and possibly some auxiliary information that is generated during the initial computation of the solution. This in turn leads us to a third stage which consists of updating the auxiliary information.

When we compare the computational complexity of an incremental algorithm to that of start-over algorithms we need to consider the complexity of all three stages of the incremental algorithm. Our objective is to design incremental algorithms that are efficient when compared to start-over algorithms.

An important aspect of incremental algorithms is the design of data structures to store the previous solution as well as some auxiliary information that is generated during the initial computation. Such data structures should provide rapid access to the necessary information for efficient updates of the solution. As we will see later on our update algorithms require fast identification of the vertices that belong to two different subtrees that are created by deleting an edge from the tree. For an inverted tree (a rooted tree where a node points towards its parent) these computations can be done in parallel in O(log n)[++] time on our model of computation (see [12]). It was shown in [8] that an inverted spanning tree can be used for parallel update of a minimum spanning tree in O(log n) time. In this paper we show that some more graph properties such as connected components and bridges can be updated in O(log n) time using an inverted spanning tree of the graph. We store connected components and bridges using inverted trees. Our algorithms ensure that these trees are maintained as inverted trees after successive updates.

Our model of computation is the unbounded parallel random access machine (PRAM). We assume that all processors have access to a common memory and that simultaneous reads from the same location are allowed but simultaneous writes to the same location are prohibited. Efficient algorithms have been developed for several graph problems on this particular model of parallel computation [4-7 and 8-11]. These algo-

[++] Throughout this paper, we use log n to denote $\lceil \log_2 n \rceil$.

rithms provide a standard with which we compare the complexity of our incremental algorithms.

The rest of the paper is organized into four sections. In Section 2 we describe some graph-theoretic preliminaries adopting the framework in [14]. In Section 3 we describe the update algorithm for connected components. The update algorithms for bridges and bridge-connected components are described in Section 4.

2. Preliminaries

In order to describe our algorithms to update graph properties we now present some graph theoretic preliminaries.

Let G=(V,E) denote a *graph* where V is a finite set of vertices and E is a set of pairs of vertices called edges. If the edges are unordered pairs then G is *undirected* else it is *directed*. Throughout this paper we assume that $V=\{1,2,...,n\}$, $|V|=n$ and $|E|=m$. We denote the undirected edge from a to b by (a,b) and the directed edge between them by $<a,b>$. We say that an undirected graph G is *connected* if for every pair of vertices u and v in V, there is a path in G joining u and v. Each connected maximal subgraph of G is called a *component* of G. An adjacency matrix A of G is an $n \times n$ Boolean matrix such that $A[u,v]=1$ if and only if $(u,v) \in E$. A *tree* is a connected undirected graph with no cycles in it. Let $T=(V', E')$ be a directed graph. T is said to have a *root* r, if $r \in V'$ and every vertex $v \in V'$ is reachable from r via a directed path. If the underlying undirected graph of T is a tree then T is called a directed tree. If the edges of T are all reversed then the resulting graph is called an *inverted* tree. An *inverted spanning tree* (IST) and an *inverted spanning forest* (ISF) are defined similarly. We denote an undirected path from vertex a to vertex b by [a-b] and directed path by $[a \rightarrow b]$. We say that vertex w is an ancestor of vertex v if w is on the path from v to the root of the tree. Let T be a directed tree with $u,v \in V'$. Then the *lowest common ancestor* (LCA(u,v)) of u and v in T is the vertex $w \in V'$ such that w is a common ancestor of u and v, and any other common ancestor of u and v in T is also an ancestor of w in T.

As we will see later on, our update algorithms require the paths from all vertices to the root in an inverted tree. Tsin and Chin [14] have described a technique due to Savage [10] to compute all such paths. For completeness we now describe their technique.

Let $T=(V',E')$ be an inverted tree with $V'=\{1,2,...,n\}$ and $|V'|=n$. Let r be the root of this tree. For a directed edge $<a,b>$ we say that vertex b is the father of vertex a.

Definition: $F:V' \rightarrow V'$ is a function such that F(i)=the father of vertex i in T for $i \neq r$ and $F(r)=r$.

The function F can be represented by a directed graph F which can be constructed from T by adding a self-loop to the root r.

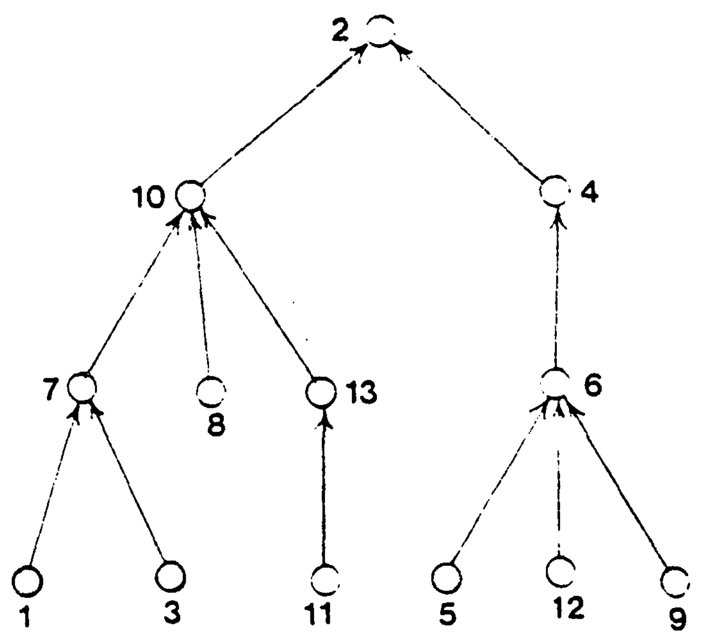
From the function F, we define F^k, $k \geq 0$ as follows.

Definition: $F^k:V' \rightarrow V'$ $(k \geq 0)$ such that $F^0(i)=i$ for all $i \in V'$ and $F^k(i)=F(F^{k-1}(i))$ for all $i \in V'$ and $k>0$.

If i is a vertex in T, $F^k(i)$ is the kth ancestor of i in the inverted tree.

Definition: For each $i \in V'$, $depth(i)=\min\{k | F^k(i)=r \text{ and } 0 \leq k < n\}$.

Lemma 2.1: Given the function F of an inverted tree, F^k can be computed in O(log n) time using $O(n^2)$ processors.



	0	1	2	3	4	5	6	7	8	9	10	11	12
1										1	7	10	2
2													2
3										3	7	10	2
4												4	2
5										5	6	4	2
6											6	4	2
7											7	10	2
8											8	10	2
9										9	6	4	2
10												10	2
11										11	13	10	2
12										12	6	4	2
13											13	10	2

Undefined entries are left blank.

Fig. 2.1

Proof: We compute the function F^k by successive compositions of functions F^i $(i<k)$ that have been determined in previous iterations. The number of ancestors of a vertex that are marked in each iteration increases by a factor of two. Since a vertex can have at most n-1 ancestors, we need O(log n) iterations to mark all ancestors. To do the i^{th}

iteration in constant time we require $2^i n$ processors. As there are log(n-1)-1 iterations, we require $O(n^2)$ processors. □

The actual computations of $F^k(i)$ $(1 \leq i \leq n, 1 \leq k < n)$ are performed in an array F^+ in which $F^+[i,k]$ contains $F^k(i)$. Once the F^+ array is computed, depth(i) $(1 \leq i \leq n)$ can be found by performing a binary search on the i^{th} row. We search for the leftmost occurrence of r. This takes log n time by assigning a processor per row. However, it can be done in constant time by assigning a processor to each element in F^+. This is done as follows. Every processor compares its element with the elements in its left and right neighbors. There is exactly one processor which does not have all the three elements identical or distinct and this processor locates the leftmost occurrence of r. The depth information is stored in a one-dimensional array D^+.

After the computations for D^+ are finished, each row of F^+ is right shifted so that all the r's except the leftmost one are eliminated. As a consequence, the rightmost column of the array contains only the root r. Fig. 2.1 illustrates an inverted tree and its array F^+ after the rows have been shifted right.

Lemma 2.2: We can compute the lowest common ancestors of all vertex pairs in the inverted tree in O(log n) time using $O(n^2)$ processors.

Proof: We make use of the array F^+ to design a parallel algorithm for finding the lowest common ancestors. For an n vertex graph there are ${}^nC_2$ (the number of unordered pairs of n elements) vertex pairs, that is $O(n^2)$ pairs. Let a and b be a vertex pair. If c is their lowest common ancestor, then row a and row b of F^+ will have identical contents for column n-1, column n-2,..., down to the column containing c. After this column the contents of rows a and b differ. As a result, to determine c, we can perform a binary search on row a and row b simultaneously in the following way. If the two entries being examined in row a and row b (in the same column) are different, the search is continued on the right half, otherwise it is continued on the left half. It takes (log n)+1 time steps to find c with one processor. Therefore we need $O(n^2)$ processors to find the lowest common ancestors of all vertex pairs. $\square$

Having obtained the lowest common ancestor we can now identify the unique path between any two vertices (passing through their lowest common ancestor). We now describe our parallel algorithms for updating connected components.

3. Connected Components

The problem of updating connected components of a graph deals with recomputing the sets of vertices, one for each component, after an incremental change has been made to the graph. The start-over algorithm for finding connected components [6] does not compute these sets directly but outputs a function $R_c:V \rightarrow V$ such that $R_c(i)$ is the representative vertex of the connected component of G to which the vertex i belongs. Incremental algorithms for updating connected components update the function R_c.

Cheston [1] in his thesis has compared several sequential algorithms for updating connected components. Even and Shiloach [3] have described a sequential algorithm for an on-line edge deletion update of connected components. Hirschberg et al. [6] have described a parallel algorithm for computing connected components on a PRAM that requires $O(\log^2 n)$ time and uses $O(n^2)$ processors.

Our algorithms for the edge update as well as vertex insertion problem require O(log n) time and use $O(n^2)$ processors. We assume that the update algorithms operate on an inverted spanning forest (ISF) for the graph. Using a technique due to Tsin and Chin [12] the start-over algorithm for finding the connected components can be modified to generate an ISF for the graph that contains one IST for each connected component of the graph. The representative vertex for each component is the root of the corresponding IST. The algorithms described in this section first update the ISF and then compute the array F^+. (See Section 2.) The last column of F^+ specifies the updated function R_c which in turn determines the new connected components.

Edge Update Algorithms

We now describe our algorithm to reconstruct the new ISF after an edge has been deleted or added into the underlying graph.

If the edge [say, (x,y)] to be deleted is not in the forest then the forest remains unchanged. On the other hand if (x,y) belongs to one of the trees in the forest then that particular tree can be reconstructed as follows.

1. *Delete the tree edge (x,y). This replaces the tree by two subtrees.*
2. Identify the vertices in each of these subtrees.
3. Find an edge (u,v) connecting them.

When we add an edge (u,v) to the graph, the forest remains unchanged if edge (u,v) connects two vertices belonging to the same tree. If (u,v) connects two different trees in the forest then (u,v) is added to the forest. The edge (u,v) and the two trees connected by it form a new tree in the forest.

Since our algorithms operate on an ISF we must now maintain the new forest as an ISF. The actual computational steps are described below. Let (x,y) be the edge to be deleted from an IST rooted at r.

1. We assume, without loss of generality, that the direction of edge (x,y) is from x to y. To delete (x,y) set $F^1(x)=x$. This creates two subtrees, one of which is rooted

at r and the other at x. This step can be done in constant time using a single processor.

2. Compute the array F^+. By Lemma 2.1 this can be done in O(log n) time using $O(n^2)$ processors. At the end of this step all vertices in the subtree rooted at r will have r in their last column in F^+ and the vertices in the subtree rooted at x will have x in their last column. We therefore can identify the vertices in the two subtrees.

3a. For each vertex i in the subtree rooted at r find an edge (i,j) such that j is in the subtree rooted at x and j is the minimum among all neighbors of i. This can be done in O(log n) time by assigning n processors to every vertex.

3b. The edge (u,v) connecting these two subtrees can now be found from the edges selected in step 3a such that v is the minimum among all j's belonging to the subtree rooted at x. If no such edge exists then deletion of edge (x,y) has disconnected the tree to which it belonged, creating two ISTs, one for each component.

4. If the edge (u,v) does exist then add it to the forest connecting the two subtrees created in step 1. In order to maintain the new tree thus formed as an inverted tree, we proceed as follows.

   Assume, without loss of generality, that u is in the subtree rooted at x and v is in the subtree rooted at r. Now orient the edge (u,v) from u to v. To do so set $F^1(u)=v$. In step 2 we found the path from vertex v to x. Reverse the directions of the edges on the directed path $[v \rightarrow x]$ in the old inverted tree. For instance, if the directed edge $<a,b>$ was on the directed path $[v \rightarrow x]$ then set $F^1(b)=a$. This path can have at most n edges and hence the reversal can be done in constant time using n processors.

5. In order to update the function R_c, compute F^+. The last column of F^+ determines the new set of connected components. This step can be done in O(log n) time using $O(n^2)$ processors.

The computational steps to update ISF when an edge (u,v) is added to the graph connecting two ISTs in the forest are the same as steps 4 and 5 of the edge deletion algorithm. The array F^+ is computed at the end of step 5. After the computation, the last column of F^+ completely determines the function R_c.

Vertex Update Algorithm

The vertex update problem involves reconstructing the ISF after a vertex has been deleted or inserted into the underlying graph. Our approach to the vertex insertion problem is a generalization of the edge insertion algorithm, where all the new edges added to the graph are incident on the new vertex. The other case of updating the ISF when a vertex is deleted from it appears difficult to handle. For instance, if one of the ISTs in the forest is in the form of a star (that is, there exists a vertex on which all the edges in the tree are incident), the deletion of such a vertex deletes all the edges in the tree. Updating the ISF then requires reconstructing that particular tree all over again and this takes $O(\log^2 n)$ time.

In order to update the ISF after a new vertex has been inserted into the underlying graph we select a new edge for each component that connects it to the new vertex. We assign directions to new edges entering the forest and re-orient some of the old edges in the forest. The computational steps are as follows.

Let z be the new vertex inserted into the graph.

1. Compute the array F^+. The last column of F^+ identifies the vertices in the connected components of the graph. This can be done in O(log n) time using $O(n^2)$ processors.

2. Identify the multiple edges incident on each component that are brought in by the vertex z. Select an edge (w,z) for each connected component such that w is the minimum among the vertices that have now become adjacent to the vertex z. This selection can be done as follows. Define a vector of length n for each connected component identified by the root of the corresponding IST. Now assign a processor to each new edge (v,z). This processor marks the v^{th} location in a vector defined for the component to which the vertex v belongs. This marking takes constant time and uses at most n processors. Now assign n processors to each vector and select a minimum vertex among the vertices that are marked. This step requires O(log n) time and uses $O(n^2)$ processors.

3. Let $r_1, r_2, \ldots, r_k$ be the roots of the k trees in the forest that are now connected to z. Let $w_1, w_2, \ldots, w_k$ be the vertices in these trees that are selected in step 2. The array F^+ computed in step 1 contains the paths from w_1 to r_1, w_2 to r_2,...,w_k to r_k. Reverse the directions of all the edges on these paths. Next orient all the edges $(w_1,z), (w_2,z), \ldots, (w_k,z)$ towards z. Thus z becomes the root of the new inverted tree formed by k trees in the forest and the vertex z. Reversal of the edges can be done in constant time using O(n) processors.

4. Compute the array F^+. Again the last column of F^+ completely determines the sets of vertices in the connected components of the new graph.

This completes the description of our algorithms to reconstruct the new ISF after a minor change has been made to the graph. This reconstruction requires O(log n) time and uses $O(n^2)$ processors.

The inverted spanning forest (tree) appears to be a very useful data structure to store the information regar ˙˙rg the connectivity of the graph. Several other graph properties can be recomputed once the ISF for a new graph is reconstructed. This enables us to develop efficient algorithms for updating these properties. For instance, we can compute the new set of fundamental cycles for the modified graph. Note that every edge not in the tree induces a fundamental cycle in the tree. In order to recompute the set of fundamental cycles we proceed as follows.

1. Reconstruct the IST. Compute the array F^+ and determine the LCA for every pair of vertices in the graph. Let u be the LCA of the vertex pair (v,w). LCA computation can be done in $O(\log n)$ time using $O(n^2)$ processors. (See Section 2.)
2. The fundamental cycle induced by the edge (v,w) consists of paths $[v \rightarrow u]$ and $[u \rightarrow w]$ in the tree and the edge (v,w). These paths can be determined using the rows for the vertices v and w in the array F^+.

In the next section we show that the IST can be used to update bridges and bridge-connected components of an undirected and connected graph.

4. Bridges and Bridge-Connected Components

In this section we describe incremental algorithms for finding the bridges of an undirected graph after a change has been made to the graph. Assume, without loss of generality, that the undirected graph is connected. Tsin and Chin [12] have described an $O(\log^2 n)$ algorithm for finding all the bridges in a connected undirected graph on a PRAM. Our update algorithms require $O(\log n)$ time and use $O(n^2)$ processors.

The algorithms presented in this section depend on a function HLCA:V→V which was first defined in [12] as follows. (H stands for highest.)

Definition: Let G=(V,E) be an undirected graph and T=(V,E$'$) be its IST and u ϵ V. HLCA(u) is a vertex nearest to the root of T such that it is an LCA(u,v), where (u,v) is an edge not in T.

The computation of the function HLCA involves numbering the vertices in T in preorder. Recently Tarjan and Vishkin [11] have described a novel algorithm to compute tree functions which include preorder numbering. Their algorithm takes O(log n) time and uses n processors. Let pmax(u) and pmin(u) denote a vertex that has maximum and minimum preorder number among the neighbors of u respectively. The functions pmax and pmin can be computed in O(log n) time using n processors per vertex, since it involves finding a maximum and a minimum of at most n numbers for each vertex. Let vertex v be pmax(u) and vertex w be pmin(u). Then HLCA(u) is either the vertex LCA(u,v) or the vertex LCA(u,w) depending upon whichever vertex is nearer to the root of T. We therefore have the following lemma.

Lemma 4.1: Let T=(V,E') be an IST of G with vertices labeled in preorder. Then HLCA(u) for all u in V can be determined in O(log n) time using O(n^2) processors.

In order to determine the bridges of a graph the following characterization of bridges is useful.

Lemma 4.2: If an edge in a graph G is a bridge then it belongs to every spanning tree of G.

Lemma 4.3: An edge (x,F^1(x)) in an IST of G is a bridge iff for every descendant i of x, its adjacent vertices are also descendants of x.

The proofs of these lemmas are obvious. Using the function HLCA and Lemma 4.3, we can easily determine that an edge $(x,F^1(x))$ is a bridge iff for every descendant i of x, $\text{depth}(\text{HLCA}(i)) > \text{depth}(F^1(x))$. Since each vertex in T can have at most n descendants, assign n processors to every edge in T. Using the results of n such depth comparisons performed for every edge, we now can determine if that edge is a bridge or not in O(log n) time using O(n) processors. We therefore have the following theorem.

Theorem 4.1: Given the HLCA of every vertex in T we can determine all the bridges in O(log n) time using $O(n^2)$ processors.

Edge Update

The best known start-over algorithm [12] for finding all bridges of an undirected graph first constructs an IST of the graph and then determines the bridges by computing the HLCA of every vertex in the IST (see Theorem 4.1). The construction of an IST takes $O(\log^2 n)$ time and the rest of the steps require O(log n) time as described earlier. Since we can update the IST of the graph in O(log n) time when an edge is deleted or added or a vertex is inserted into the underlying graph, we therefore can determine the bridges for the new graph in O(log n) time. However, such a straightforward approach would test all the edges every time the graph undergoes a change. We now describe another approach that requires testing of fewer edges. This approach is based on the following lemma whose proof is straightforward.

Lemma 4.4: An edge in a spanning tree of a graph is a bridge iff it does not lie on any fundamental cycle.

Now consider the case when an edge (u,v) is added to the graph. The edge (u,v) induces a fundamental cycle in the IST which can be determined using the array F^+ and LCA(u,v). (See Section 3.) All the bridges lying on this cycle now become non-bridges. The computation of F^+ and LCA for every pair of vertices requires O(log n) time and uses $O(n^2)$ processors. The second step (finding the bridges lying on the cycles and marking them as non-bridges) can be done in constant time and needs at most n processors. The IST of the graph remains unchanged.

The other case of determining bridges after an edge has been deleted is handled as follows. There are two subcases to be considered. In the first subcase we assume that the deleted edge (x,y) was not in the tree. We then need to recompute the HLCA of all the vertices lying on the cycle induced by edge (x,y). All the edges lying on this cycle need to be tested as to whether they have become bridges. As there are at most n vertices and (n-1) tree edges lying on such a cycle, we can do this in O(log n) time using $O(n^2)$ processors. (See Lemmas 4.1 and 4.3.)

For the second subcase, assume that the deleted edge was in the IST. Deletion of the edge (x,y) creates two subtrees. Update the IST using the technique described in Section 3. If (x,y) was a bridge then the IST becomes an ISF and the rest of the bridges are not affected. On the other hand if (x,y) was not a bridge then let (u,v) be an edge selected during an update step that connects the subtrees. Now this case is equivalent to the previous subcase if we treat (x,y) as a non-tree edge that was deleted. We again need to test the edges lying on the cycle induced by the edge (x,y) in the new IST.

This completes the description of the edge update algorithms for determining all bridges of the modified graph.

Vertex Insertion

The case of vertex insertion is handled as an extension of the edge insertion problem.

Let z be the new vertex that is added to the graph. Construct the new IST according to the vertex insertion algorithm described in Section 3. If in the new graph z has only one edge incident on it, then that edge is in the IST and it is a bridge. All the other bridges remain unchanged. Suppose z has more than one incident edge in the new graph. There can be at most (n-1) such edges which are incident on z but are not in the tree. (This is because we just selected one representative edge that connects z to the old IST.) Compute the array F^+ and the function LCA for the new IST of the graph. These (n-1) edges induce (n-1) cycles in the IST. All bridges lying on these cycles become non-bridges. Since there are at most (n-1) edges on each cycle that are to be tested, we need $O(n^2)$ processors to mark the bridges as non-bridges in constant time. However, more than one processor may mark the same edge as a non-bridge, resulting in a write conflict (disallowed by our model of computation). Such a write conflict can be avoided using a technique due to Hirschberg [5] which marks all the bridges lying on these cycles as non-bridges in O(log n) time using $O(n^2)$ processors.

Bridge-Connected Components

Once we have updated the bridges of the modified graph, the bridge-connected components can be determined as follows.

1. Delete all the bridges from the IST. This creates a forest of ISTs, one for each component.

2. Compute the array F^+. This takes O(log n) time and uses O(n^2) processors. The last column of F^+ determines the new bridge-connected components of the modified graph.

5. Conclusions

Incremental graph algorithms deal with recomputing properties of a graph after an incremental change has been made to the graph. In this paper we have described parallel algorithms to update connected components and bridges of an undirected graph after an edge has been inserted or deleted or a new vertex has been inserted in the underlying graph. Our algorithms require O(log n) time and use O(n^2) processors and therefore are efficient when compared to the start-over algorithms. We have shown that an inverted tree is a useful data structure for developing algorithms to update properties of an undirected graph. It would be interesting to explore the applicability of this data structure to a variety of other graph problems.

References

[1] G. Cheston, "Incremental Algorithms in Graph Theory", TR 91, Dept. of Computer Science, Univ. of Toronto, Toronto (1976).

[2] N. Christofides, "Graph Theory: An Algorithmic Approach", Academic Press, New York (1975).

[3] S. Even and Y. Shiloach, "An On-line Edge Deletion Problem", *J. ACM*, 28 (1982), pp 1-4.

[4] D. Hirschberg, "Parallel Algorithms for the Transitive Closure and the Connected Component Problems", *Proc. of Eighth ACM Symposium on Theory of Computing* (1976), pp 55-57.

[5] D. Hirschberg, "Fast Parallel Sorting Algorithms", *Comm. ACM*, 21 (1978), pp 657-661.

[6] D. Hirschberg, A. K. Chandra and D. V. Sarwate, "Computing Connected Components on Parallel Computers", *Comm. ACM*, 22 (1979), pp 461-464.

[7] M. J. Quinn and N. Deo, "Parallel Graph Algorithms", *ACM Comp. Surveys*, 16 (1984), pp 319-348.

[8] I. V. Ramakrishnan and S.Pawagi, " Parallel Update of Minimum Spanning Trees in Logarithmic Time", TR 1452, Dept. of Computer Science, Univ. of Maryland, College Park (1984).

[9] C. Savage, "Parallel Algorithms for Some Graph Problems", TR-784, Dept. of Mathematics, Univ. of Illinois, Urbana (1977).

[10] C. Savage and J. Ja'Ja', "Fast Efficient Parallel Algorithms for Some Graph Problems", *SIAM J. Comp.*, 10 (1981), pp 682-691.

[11] R. E. Tarjan and U. Vishkin, "Finding Biconnected Components and Computing Tree Functions in Logarithmic Parallel Time", *Proc. of Twenty-third Annual FOCS Symposium* (1984), pp 12-20.

[12] Y. Tsin and F. Chin, "Efficient Parallel Algorithms for a Class of Graph Theoretic Problems", *SIAM J. Comp.*, 14 (1984), pp 580-599.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A160 135

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CAR-TR-117 CS-TR-1492	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFOSR-TR- ~~N/A~~ [illegible] 822
6a. NAME OF PERFORMING ORGANIZATION University of Maryland 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Air Force Office of Scientific Res.
6c. ADDRESS (City, State and ZIP Code) Center for Automation Research College Park, MD 20742	7b. ADDRESS (City, State and ZIP Code) Bolling Air Force Base Washington, DC 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION - 8b. OFFICE SYMBOL (If applicable) -	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F49620-83-C-0082
8c. ADDRESS (City, State and ZIP Code) -	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: 61102F PROJECT NO.: 2304 TASK NO.: K2 WORK UNIT NO:
11. TITLE (Include Security Classification) On Using Inverted Trees for Updating Graph Properties	
12. PERSONAL AUTHOR(S) Shaunak Pawagi and I.V. Ramakrishnan	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM ______ TO N/A	14. DATE OF REPORT (Yr., Mo., Day) May 1985 15. PAGE COUNT 20
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Fast parallel algorithms are presented for updating connected components and bridges of an undirected graph when a minor change has been made to the graph, such as addition or deletion of vertices and edges. The machine model used is a parallel random access machine which allows simultaneous reads but prohibits simultaneous writes into the same memory location. The algorithms described in this paper require $O(\log n)$ time and use $O(n^2)$ processors. These algorithms are efficient when compared to previously known algorithms for finding connected components and bridges that require $O(\log^2 n)$ time and use $O(n^2)$ processors. The previous solution is maintained using an inverted tree (a rooted tree where a node points towards its parent) and after a minor change the new solution is rapidly computed from this tree.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL [illegible]	22b. TELEPHONE NUMBER (Include Area Code) (202) 767-4940 22c. OFFICE SYMBOL [illegible]

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE